# Optimizing Complex Extraction Programs over Evolving Text Data

Fei Chen[1], Byron J. Gao[2], AnHai Doan[1], Jun Yang[3], Raghu Ramakrishnan[4,1]

[1]University of Wisconsin-Madison, [2]Texas State University-San Marcos,
[3]Duke University, [4]Yahoo! Research

## ABSTRACT

Most information extraction (IE) approaches have considered only static text corpora, over which we apply IE only once. Many real-world text corpora however are dynamic. They evolve over time, and so to keep extracted information up to date we often must apply IE repeatedly, to consecutive corpus snapshots. Applying IE *from scratch* to each snapshot can take a lot of time. To avoid doing this, we have recently developed Cyclex, a system that recycles previous IE results to speed up IE over subsequent corpus snapshots. Cyclex clearly demonstrated the promise of the recycling idea. The work itself however is limited in that it considers only IE programs that contain *a single IE "blackbox."* In practice, many IE programs are far more complex, containing *multiple IE blackboxes* connected in a compositional "workflow."

In this paper, we present Delex, a system that removes the above limitation. First we identify many difficult challenges raised by Delex, including modeling complex IE programs for recycling purposes, implementing the recycling process efficiently, and searching for an optimal execution plan in a vast plan space with different recycling alternatives. Next we describe our solutions to these challenges. Finally, we describe extensive experiments with both rule-based and learning-based IE programs over two real-world data sets, which demonstrate the utility of our approach.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

## General Terms

Design, Experimentation, Performance

## Keywords

Information Extraction, Optimization, Evolving Text


*SIGMOD'09,* June 29–July 2, 2009, Providence, Rhode Island, USA.


## 1. INTRODUCTION

Over the past decade, the problem of information extraction (IE) has attracted significant attention. Given a *text corpus* (e.g., a collection of emails, Web pages, etc.), much progress has been made on developing solutions for extracting information from the corpus effectively [10, 2, 25, 15] (see also [26, 14] for the latest survey and special issue).

Most of these IE solutions however have considered only *static* text corpora, over which we typically have to apply IE only *once* [26]. In practice, text corpora often are *dynamic*, in that documents are added, deleted, and modified. Thus, to keep extracted information up to date, we often must apply IE *repeatedly*, to consecutive corpus snapshots. Consider for example DBLife [13, 12], a structured portal for the database community. DBLife operates over a text corpus of 10,000+ URLs. Each day it re-crawls these URLs to generate a 120+ MB corpus snapshot, and then applies IE to this snapshot to find the latest community information (e.g., which database researchers have been mentioned where in the past 24 hours). As another example, the Impliance project at IBM Almaden seeks to build a system that manages all information within an enterprise [4, 3]. This system must regularly re-crawl and then re-apply IE to the enterprise intranet, to infer the latest information. Recent efforts (e.g., [22, 29, 9], *freebase.com*) have also focused on converting *Wikipedia* and its wiki "siblings" into structured databases, and hence must regularly re-crawl and re-extract information. See [8, 16, 5] for other examples of dynamic text corpora.

Despite the pervasiveness of dynamic text corpora, no satisfactory solution has been proposed for IE over them. Given such a corpus, the common solution today is to apply IE to each corpus snapshot *in isolation, from scratch.* This solution is simple, but highly inefficient, with limited applicability. For example, in DBLife re-applying IE from scratch takes 8+ hours each day, leaving little time for higher-level data analysis. As another example, time-sensitive applications (e.g., stock, auction, intelligence analysis) often want to refresh information quickly, by re-crawling and re-extracting, say, every 30 minutes. In such cases applying IE from scratch is inapplicable if it already takes more than 30 minutes. Finally, this solution is ill-suited for *interactive debugging* of IE applications over dynamic corpora, because such debugging often requires applying IE repeatedly to multiple corpus snapshots. Thus, given the growing need for IE over dynamic text corpora, it has now become crucial to develop efficient IE solutions for these settings.

In response, we have recently developed Cyclex, a solu-

tion for IE over evolving text data [6]. The key idea underlying Cyclex is to recycle previous IE results, given that consecutive snapshots of a text corpus often contain much overlapping content. For example, suppose that a snapshot contains the text fragment "The Cimple project will meet in CS 105 at 3 pm", from which we have extracted "CS 105" as a room number. Then under certain conditions (see Section 3), if a subsequent corpus snapshot also contains this text fragment, we can immediately conclude that "CS 105" is a room number, without having to run (often expensive) IE operations.

The Cyclex work clearly established that recycling IE results for evolving text corpora is highly promising. The work itself however suffers from a major limitation: it considers only IE programs that contain a *single IE "blackbox."* Real-world IE programs, in contrast, often contain *multiple IE blackboxes* connected in a compositional "workflow." As a simple example, a program to extract meetings may employ an IE blackbox to extract locations (e.g., "CS 105"), another IE blackbox to extract times (e.g., "3 pm"), then pairs locations and times and keeps only those that are within 20 tokens of each other (thus producing ("CS 105", "3 pm") as a meeting instance in this case).

The IE blackboxes are either off-the-shelf (e.g., downloaded from public domains or purchased commercially) or hand-coded (e.g., in Perl or Java), and they are typically "stitched together" using a procedural (e.g., Perl) or declarative language (e.g., UIMA, Gate, xlog [17, 11, 28]). Such multi-blackbox IE programs could be quite complex, for example, 45+ blackboxes stacked in five levels in DBLife, and 25+ blackboxes stacked in seven levels in Avatar [15]. Since Cyclex is not aware of the compositional nature of such IE programs (effectively treating the whole program as a large blackbox), its utility is severely limited in such settings.

To remove this limitation, in this paper we describe Delex, a solution for effectively executing multi-blackbox IE programs over evolving text data. Like Cyclex, Delex aims at recycling IE results. However, compared with Cyclex, developing Delex is fundamentally much harder, for three reasons.

First, since the target IE programs for Delex are multi-blackbox and compositional, we face many new and difficult problems. For example, how should we represent multi-blackbox IE programs, e.g., how to stitch together IE blackboxes? How to translate such programs into execution plans? At which level should we reuse such plans? We show for instance that reusing at the level of each IE blackbox (i.e., storing its input/output for subsequent reuse), like Cyclex does, is suboptimal in the compositional setting. Once we have decided on the level of reuse, what kind of data should we capture and store for subsequent reuse? Can we reuse across IE blackboxes? These are examples of problems that Cyclex did not face.

Second, since a target IE program now consists of many blackboxes, all attempting reuse at the same time, Delex faces a far harder challenge of coordinating their execution and reuse to ensure efficient movement of large quantities of data between disk and memory. In contrast, Cyclex only had to worry about the efficient execution of a single IE blackbox.

Finally, the main optimization challenge in Cyclex is to decide which *matcher* to assign to the sole IE blackbox. (A matcher encodes a way to find overlapping text regions between the current corpus snapshot and the past ones, for the purpose of recycling IE results; see Section 3.) Thus, the Cyclex plan space is bounded by the (relatively small) number of matchers. In contrast, Delex can assign to each IE blackbox in the program a different matcher. Hence, it must search a blown-up plan space (exponential in the number of blackboxes). To exacerbate the search problem, optimization in this case is "non-decomposable;" i.e., we cannot just optimize parts of a plan, then glue the parts together to obtain an optimized whole.

In this paper we develop solutions to the above challenges. As we will show in Sections 4-6, our solutions heavily exploit properties of text and information extraction. Overall, we make the following contributions.

- We establish that it is possible to exploit work done by previous IE runs to significantly speed up complex, multi-blackbox IE programs over evolving text. As far as we know, Delex is the first solution to this important problem.
- We show how to instrument such complex IE programs for reuse. We decide the level of reuse, what to capture for reuse, and how to efficiently store the captured results.
- We show how to reuse efficiently while executing an instrumented IE program. The solution involves complex coordination among multiple IE blackboxes on extraction and reuse, across all data pages in a corpus snapshot as well as across IE blackboxes.
- We show how to estimate cost of each plan (that considers a reuse alternative), and how to efficiently search a vast plan space for a good one.
- We conduct extensive experiments with both rule-based and learning-based IE programs over two real-world data sets to demonstrate the utility of our approach. We show in particular that Delex can cut Cyclex's time by as much as 71%.

## 2. RELATED WORK

The problem of information extraction has received much attention [26, 14, 2, 15, 10]. Numerous rule-based extractors (e.g., those relying on regular expressions or dictionaries [12, 25]) and learning-based extractors (e.g., those employing classifiers, CRF models [30, 27, 26]) have been developed. Delex can handle both types of extractors (as we show in the experiments).

Much work has tried to improve the accuracy and runtime of these extractors [26]. But recent work has also considered how to combine and manage such extractors in large-scale IE applications [2, 15, 1]. Our work fits into this emerging direction.

In terms of IE over *evolving text data*, Cyclex [6] is the closest work to ours. But Cyclex is limited in that it considers only IE programs that contain a single IE blackbox, as we have discussed. [23] also considers evolving text data, but in different problem contexts. They focus on how to incrementally update an inverted index, as the indexed Web pages change.

Recent work [31, 19] has also exploited *overlapping text data*, but again in different problem contexts. These works observe that document collections often contain overlapping text. They then consider how to exploit such overlap to "compress" the inverted indexes over these documents, and

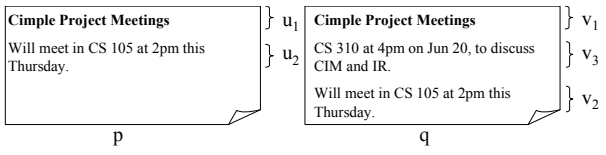


**Figure 1: Two pages $p$ and $q$ of the same URL, retrieved at different times. A matcher has found that regions $u_1$ and $u_2$ of page $p$ match regions $v_1$ and $v_2$ of page $q$, respectively.**

how to answer queries efficiently over such compressed indexes. In contrast, we exploit the IE results over the overlapping text regions to reduce the overall extraction time.

Our work is also related to *incremental view maintenance* [18] – namely, if changes to the input of a dataflow program are small, then incrementally computing changes to the result can be more efficient than recomputing the dataflow from scratch. But the works differ in many important ways. First, our inputs are text documents instead of tables. Most work on view maintenance assumes that changes to the inputs (base tables) are readily available (e.g., from database logs), while we also face the challenge of how to characterize and efficiently detect portions of the input texts that remain unchanged. Most importantly, view maintenance only needs to consider a handful of standard operators with well-defined semantics. In contrast, we must deal with arbitrary IE blackboxes.

Finally, optimizing IE programs and developing IE-centric cost models have also been considered in several recent papers [28, 20, 21]. These efforts however have considered only static corpus contexts, not dynamic ones as we do in this paper.

## 3. PROBLEM DEFINITION

**Data Pages, Extractors, and Mentions:** We now briefly describe Cyclex [6], then build on it to define the problem considered in this paper. Cyclex considers an application that crawls a set of data sources at regular intervals to retrieve *data pages*. We refer to the set of data pages retrieved at time $i$ as $P_i$, the *$i$-th snapshot* of the evolving text corpus.

The goal of Cyclex is to extract a target relation $R$ from the data pages. To do so, it employs an extractor $E$, which is typically a learning-based program or a set of extraction rules encoded in, say, a Perl script [15]. Formally:

DEFINITION 1 (EXTRACTOR). *An* extractor $E : p \rightarrow R(a_1, \ldots, a_n)$ *extracts mentions of relation $R$ from page $p$. A* mention *of $R$ is a tuple $(m_1, \ldots, m_n)$, such that $m_i$ is either a* mention *of attribute $a_i$ (i.e., a string in page $p$ that provides a value for $a_i$) or* nil *(if $E$ did not find a value for $a_i$).*

For example, from page $p$ in Figure 1, an extractor $E$ may extract mention ("CS 105","2pm") of the target relation MEETING(room,time), where "CS 105" and "2pm" are mentions of attributes room and time, respectively.

As Definition 1 suggests, we assume that $E$ extracts mentions from *each data page in isolation*. Such per-page extractors are pervasive (e.g., constituting 94% of extractors in the current DBLife [15, 28]). Hence, we currently consider only such extractors, leaving multi-page extractors (e.g., those that extract mentions spanning multiple pages) for future work.

**The Cyclex Problem and Solution:** Given the above setting, Cyclex then exploits IE results over past corpus snapshots $P_1, \ldots, P_n$ to speed up IE over the current snapshot $P_{n+1}$. The following example illustrates this idea.

EXAMPLE 1. *Figure 1 show two pages $p$ and $q$ with the same URL, but obtained in consecutive snapshots. Suppose that extractor $E$ has extracted mention ("CS 105", "2pm") from page $p$. When applying $E$ to $q$,* Cyclex *attempts to recycle the above result. To do so, it first "matches" $q$ with $p$ to find overlapping text regions. Next, it copies over mentions found in the overlapping regions (regions $u_1$ and $u_2$ of page $p$ in Figure 1, which match regions $v_1$ and $v_2$ of page $q$, respectively), and then applies $E$ only to the non-overlapping portion of $q$ (region $v_3$ in this case).*

To realize the above idea, Cyclex addresses a series of challenges. First, it develops a set of "matchers," which find overlapping regions of two input pages. These matchers trade off the completeness of result for running time, so each of them may be appropriate under different circumstances.

Second, it turns out that we cannot simply copy over the mentions in overlapping regions. To see why, suppose a *hypothetical* extractor $E$ extracts meetings only if a page has fewer than five lines (otherwise $E$ produces no meetings). Then, none of the mentions of a four-line page $p$ can be copied over to a six-line page $q$, even if the text in $p$ is fully contained in $q$. To address this problem, a key idea behind Cyclex is to model certain extractor properties, then exploit them to reuse mentions and to guarantee the correctness of the reuse.

Specifically, Cyclex defines two properties: *scope* and *context*. An extractor $E$ has scope $\alpha$ iff all of the mentions that it extracts do not exceed $\alpha$ in length. Formally:

DEFINITION 2 (EXTRACTOR SCOPE). *Let $s.start$ and $s.end$ be the start and end character positions of a string $s$ in a page $p$. We say an extractor $E$ has* scope $\alpha$ *iff for any mention $m = (m_1, \ldots, m_n)$ produced by $E$, $(\max_i m_i.end - \min_i m_i.start) < \alpha$, where $m_i.start$ and $m_i.end$ are the start and end character positions of attribute mention $m_i$ in page $p$.*

We say that extractor $E$ has *context* $\beta$ iff whether it extracts any mention $m$ depends only on the small "context windows" of size $\beta$ to both sides of $m$. Formally:

DEFINITION 3 (EXTRACTOR CONTEXT). *The $\beta$-context of mention $m$ in page $p$ is the string $p[(m.start-\beta)..(m.end+\beta)]$, i.e., the string of $m$ being extended on both sides by $\beta$ characters. We say extractor $E$ has* context $\beta$ *iff for any $m$ and $p'$ obtained by perturbing the text of $p$ outside the $\beta$-context of $m$, applying $E$ to $p'$ still produces $m$ as a mention.*

In the worst case, the scope and context of an extractor $E$ can be set to be the length of the longest page. In practice, however, they often can be set to far smaller values with some knowledge of how $E$ works (e.g., by the developer, or anyone with access to $E$'s code), as illustrated by the following examples:

EXAMPLE 2. *To extract (room,time), suppose $E$ always extracts all room and time mentions, then pairs them and keeps only those that are spanning at most 100 characters. Then $E$'s scope can be set to 100.*

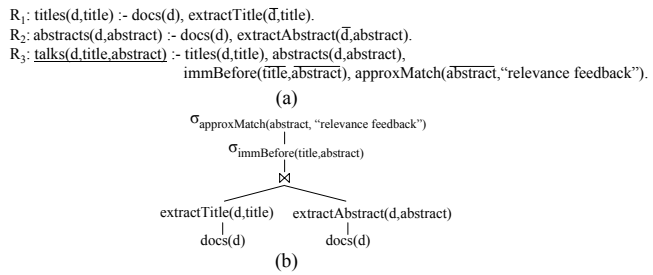


**Figure 2: (a) A multi-blackbox IE program $\mathcal{P}$ in xlog, and (b) an execution plan for $\mathcal{P}$.**

EXAMPLE 3. *Suppose $E$ produces string $X$ as a topic if $X$ matches a pre-defined word (e.g., "IR") and the word "discuss" occurs within a 30-character distance, either to the left or to the right of $X$. Then the context of $X$ can be set to 30. That is, once $E$ has extracted $X$ as a topic, then no matter how we perturb the text outside a 30-character window of $X$ (on both sides), $E$ would still recognize $X$ as a valid topic mention.*

The experiment section describes more examples of setting the scope and context.

Using scope $\alpha$ and context $\beta$, Cyclex then shows how to "safely" copy the mentions from past IE results [6]. The smaller the values of $\alpha$ and $\beta$, the more IE results we can "safely" reuse [6].

In the next step, since text corpora can be quite large (e.g., tens of thousands or millions of pages), Cyclex develops a solution that efficiently interleaves the steps of page matching, reusing, and re-extracting, over a large amount of disk-resident data.

Finally, addressing the above challenges results in a space of execution plans, where the plans differ mainly on the page matcher employed. Thus, in the final step, Cyclex develops a cost model and uses it to select the optimal plan. The cost model is extraction-specific in that it models the rate of change of the text corpus, the running time, and the result size of extractors and matchers, among other factors.

**The Generality of Our IE Model:** It is important to emphasize that the IE model defined above is quite general. First, Definition 1 does not limit the type of extractor "blackboxes" in our model. Such extractor blackboxes can be either rule-based (e.g., those that rely on regular expressions or dictionaries) or learning-based (e.g., those that use classifiers or learning models such as CRFs, Hidden Markov Models). In fact, our experiments in Section 8 show that Delex can efficiently handle both types of extractors.

Second, so far we have defined extractor scope and context at the character level (see Definitions 2-3), and in this paper, for ease of exposition, we will limit our discussion to only the character level. However, Cyclex and Delex can be easily generalized to work with scope/context at higher-granularity levels (e.g., word, sentence, paragraph), should that be more appropriate for the target extractors.

**Compositional, Multi-Blackbox IE Programs:** As discussed in Section 1, Cyclex has clearly demonstrated the potential of recycling IE. However, it handles only single-blackbox IE programs, which severely limits its applicability. Thus, in this paper, we build on Cyclex to develop an efficient solution for multi-blackbox IE programs.

To do so, we must first decide how to represent such programs. Many possible representations exist (e.g., [17, 11, 28]). As a first step, in this paper we will use xlog [28], a recently developed declarative IE representation. Extending our work to other IE representations is a subject for future research.

We now briefly describe xlog (see [28] for a detailed discussion). xlog is a Datalog variant with embedded procedural predicates. Like Datalog, each xlog program consists of multiple *rules* $p :- q_1, \ldots, q_n$, where the $p$ and $q_i$ are predicates. For example, Figure 2.a shows an xlog program $\mathcal{P}$ with three rules $R_1$, $R_2$, and $R_3$, which extract talk titles and abstracts from seminar announcement pages. Currently xlog does not yet support negation or recursion.

xlog predicates can be intensional or extensional, as in Datalog, but can also be *procedural*. A procedural predicate, or *p-predicate* for short, $q(\overline{a_1}, \ldots, \overline{a_n}, b_1, \ldots, b_m)$ is associated with a procedure $g$ (e.g., written in Java or Perl) that takes as input a tuple $(a_1, \ldots, a_n)$ and produces as output tuples of the form $(a_1, \ldots, a_n, b_1, \ldots, b_m)$. For example, $extractTitle(\overline{d}, title)$ is a p-predicate in $\mathcal{P}$ that takes a document $d$ and returns a set of tuples $(d, title)$, where *title* is a talk title appearing in $d$. We define p-functions similarly. We single out a special type of p-predicate that we call IE predicate, defined as:

DEFINITION 4 (IE PREDICATE). *An IE predicate $q$ extracts one or more output text spans from a single input span. Formally, $q$ is a p-predicate $q(\overline{a_1}, \ldots, \overline{a_n}, b_1, \ldots, b_m)$, where there exist $i$ and $j$ such that (a) $a_i$ is either a document or a text span variable, (b) $b_j$ is a span variable, and (c) for any output tuple $(u_1, \ldots, u_n, v_1, \ldots, v_m)$, $u_i$ contains $v_j$ (i.e., $q$ extracts span $v_j$ from span $u_i$).*

In Figure 2.a, $extractTitle(\overline{d}, title)$ is an IE predicate that extracts *title* span from document $d$. The p-predicate $extractAbstract(\overline{d}, abstract)$ is another IE predicate, whereas $immBefore(\overline{title}, \overline{abstract})$ (a p-predicate that evaluates to true if *title* occurs immediately before *abstract*) is not.

Thus, an xlog program cleanly encapsulates multiple IE blackboxes using IE predicates, and then stitches them together using Datalog. To execute such a program, we must translate (and possibly optimize) it to obtain an *execution plan* that mixes relational operators with blackbox procedures. Figure 2.b shows a possible execution plan $T$ for program $\mathcal{P}$ in Figure 2.a. $T$ extracts all titles and abstracts from $d$, and keeps only those (title, abstract) pairs where the title occurs immediately before the abstract. Finally, $T$ retains only talks whose abstracts contain the phrase "relevance feedback" (allowing for misspelling and synonym matching).

**Problem Definition:** We are now in a position to define the problem considered in this paper.

PROBLEM DEFINITION *Let $P_1, \ldots, P_n$ be consecutive snapshots of a text corpus, $\mathcal{P}$ be an IE program written in xlog, $E_1, \ldots, E_m$ be the IE blackboxes (i.e., IE predicates) in $\mathcal{P}$, and $(\alpha_1, \beta_1), \ldots, (\alpha_m, \beta_m)$ be the estimated scopes and contexts for the blackboxes, respectively. Develop a solution to execute $\mathcal{P}$ over corpus snapshot $P_{n+1}$ with minimal cost, by reusing extraction results over $P_1, \ldots, P_n$.*

To address this problem, a simple solution is to detect identical pages, then reuse IE results on those. This *reuse-at-page-level* solution however provides only limited reuse op-

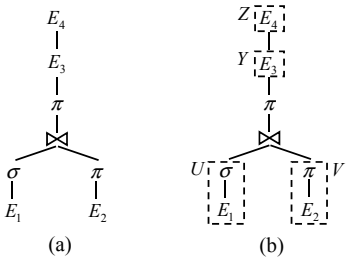


**Figure 3: (a) An execution tree $T$, and (b) IE units of $T$.**

portunities, and does not work well when the text corpus changes frequently.

Another solution is to apply Cyclex to the whole program $\mathcal{P}$, effectively treating it as a single IE blackbox. We however found that this *reuse-at-whole-program-level* solution does not work well either (see Section 8). The main reason is that estimating "tight" $\alpha$ and $\beta$ for the whole IE program $\mathcal{P}$ is very difficult. Whether we do so directly, by analyzing the behavior of $\mathcal{P}$ (which tends to be a large and complex program), or indirectly, by using the $(\alpha_i, \beta_i)$ of its component blackboxes, we often end up with large $\alpha$ and $\beta$, which limit reuse opportunities.

These problems suggest that we should try to reuse at a finer granularity: *regions in a page* instead of *whole page*, and at a finer level: *program components* instead of *whole program*. The Delex solution captures this intuition. In the rest of the paper we describe Delex in detail.

# 4. CAPTURING IE RESULTS

We will explain Delex in a bottom-up fashion. Let $T$ be an execution plan of the target IE program $\mathcal{P}$ (see Problem Definition). In this section we consider what to capture for reuse, when executing $T$ on a corpus snapshot $P_n$.

Section 5 then discusses how to reuse the captured result when executing $T$ on $P_{n+1}$. Section 6 describes how to select such a plan $T$ in a cost-based fashion. Section 7 puts all of these together and describes the end-to-end Delex solution.

In what follows we describe how to decide on the level of reuse, what to capture, and how to store the captured results, when executing $T$ on snapshot $P_n$.

**Level of Reuse:** Recall that we want to reuse at the granularity of program components, instead of the whole program. The question is *which* components. A natural choice would be the individual IE blackboxes. For example, given the execution tree[1] $T$ in Figure 3.a, the four IE blackboxes $E_1, \ldots, E_4$ would become "reuse units," whose input and output would be captured for subsequent reuse.

Reusing at the IE-blackbox level however turns out to be suboptimal. To explain, consider for instance blackbox $E_1$ (Figure 3.a), and let $\sigma(E_1)$ denote the edge of $T$ that applies the selection operator $\sigma$ to the output of $E_1$. Instead of storing the output of $E_1$, we can store that of $\sigma(E_1)$. Doing so does not affect reuse (as we will see below), but is better in two ways. First, it would incur less storage space, because $\sigma(E_1)$ often produces far fewer output tuples than $E_1$. Second, less storage space in turn reduces the time of writing to disk (while executing $T$ on $P_n$) and reading from disk (for reuse, while executing $T$ on $P_{n+1}$). Consequently, we reuse at the level of IE *units*, defined as

[1]In the rest of the paper we will use "tree," "execution tree," and "execution plan" interchangeably.

DEFINITION 5 (IE UNIT). *Let $X = N_1 \leftarrow N_2 \leftarrow \cdots \leftarrow N_k$ denote a path on tree $T$ that applies $N_{k-1}$ to $N_k$, $N_{k-2}$ to $N_{k-1}$, and so on. We say $X$ is an* IE unit *of $T$ iff (a) $N_k$ is an IE blackbox, (b) $N_1, \ldots, N_{k-1}$ are relational operators $\sigma$ and $\pi$, and (c) $X$ is maximal in that no other path satisfying (a) and (b) contains $X$.*

For example, tree $T$ in Figure 3.a consists of four IE units $U, V, Y$, and $Z$, as shown in Figure 3.b.

In essence, each IE unit can be viewed as a generalized IE blackbox, with similar notions of scope $\alpha$ and context $\beta$. In this setting, it is easy to prove that we can set the $(\alpha, \beta)$ of an IE unit $N_1 \leftarrow N_2 \leftarrow \cdots \leftarrow N_k$ to be exactly those of the IE blackbox $N_k$. This property is desirable and explains why we do not include join operator $\bowtie$ in the definition of IE unit: doing so would prevent us from guaranteeing the above "wholesale transfer" of $(\alpha, \beta)$ values.

**IE Results to Capture:** Next we consider what to capture for each IE unit $U$ of tree $T$. Conceptually, each such unit $U$ (which is an IE blackbox $E$ augmented with $\sigma$ and $\pi$ operators, whenever possible) can be viewed as extracting a set of mentions from a text region of a document. Formally, we can write $U : (did, s, e, c) \rightarrow \{(did, m, c')\}$, where

- *did* is the ID of a document $d$,
- $s$ and $e$ are the start and end positions of a text region $S$ in $d$,
- $c$ denotes the rest of the input parameter values (see the example below),
- $m$ denotes a mention (of a target relation) extracted from text region $S$, and
- $c'$ denotes the rest of the output values.

EXAMPLE 4. *Consider a hypothetical IE unit $\sigma_{allcap(title)}(extractTitle(\overline{d}, \overline{maxlength}, title, numtitles))$, which extracts all titles not exceeding maxlength from document d, selects only those in all capital letters, and outputs them as well as the number of such titles.*

*Here, for the input tuple, did is the ID of document d, s and e are the positions of the first and last characters of d (because text region S is the entire document d), and c denotes maxlength. For the output tuple, m is an extracted title, and $c'$ denotes numtitles.*

In order to reuse the results of $U$ later, at the minimum we should record all mentions $m$ produced by $U$ (recall that given an input tuple $(did, s, e, c)$, $U$ produces as output a set of tuples $(did, m, c')$). Then, whenever we want to apply $U$ to a region $S$ in a page $p$, we can just copy over all mentions of a region $S'$ in some page $q$ in a past snapshot, which we have recorded when applying $U$ to $S'$, provided that $S$ matches $S'$ and that it is safe to copy the mentions (see Section 3).

This is indeed what Cyclex does. In the Delex context, however, it turns out that since we employ multiple IE blackboxes that can be "stacked" on top of one another, we must record more information to guarantee correct reuse, as the following example illustrates.

EXAMPLE 5. *Consider a page $p$ = "Midwest DB Courses: CS764 (Wisc), CS511 (Illinois)". Suppose we have applied an IE unit $V$ to $p$ to remove the headline (by ignoring all text before ":"), and then applied another IE unit $U$ to the rest of the page to extract locations "Wisc" and "Illinois".*

*Suppose the next day the page is modified into $p'$ = "Midwest DB Courses This Year CS764 (Wisc), CS511 (Illinois)", where character ":" has been omitted (and some new text has been added). Consequently, $V$ does not remove anything from $p'$, and $p'$ ends up sharing the region $S$ = "Midwest DB Courses" with $p$. Thus, when applying $U$ to $p'$, we will attempt to copy over mentions found in this region. Since no such mention was recorded, however, we will conclude that applying $U$ to region $S$ in $p'$ produces no mention. This conclusion is incorrect, since "Midwest" is a valid location mention in $S$.*

*The problem is that no mention has been recorded in region $S$ for $U$ and $p$, not because $U$ failed to extract any such mentions from $S$, but rather because $U$ has never been applied to $S$. $U$ can only take as input whichever regions $V$ outputs, and $V$ did not output $S$ when it operated on $p$.*

Thus, we must record not only the previously extracted mentions, but also the text regions that an IE unit has operated over. Specifically, for an IE unit $U : (did, s, e, c) \rightarrow \{(did, m, c')\}$, we record all pairs $(s, e)$ and the mentions $m$ associated with those. It is easy to see that we must record $c$ as well, for otherwise we do not know the exact conditions under which a mention $m$ was produced, and hence cannot recycle it appropriately.

**Storing Captured IE Results:** We now describe how to store the above intermediate results while executing tree $T$ on a corpus snapshot $P_n$. Our goal is to produce, at the end of the run on $P_n$, two *reuse files* $I_U^n$ and $O_U^n$ for each IE unit $U$ in tree $T$.

During the run, whenever $U$ takes as input a tuple $(did, s, e, c)$, we append a tuple $(tid, did, s, e, c)$, where $tid$ is a tuple ID (unique within $I_U^n$), to $I_U^n$, to capture the region that $U$ operates on. Whenever $U$ produces as output a tuple $(did, m, c')$, we append a tuple $(tid, itid, m, c')$ to $O_U^n$, to capture the mentions extracted by $U$. Here, $tid$ is a tuple ID (unique within $O_U^n$), and $itid$ is the ID of the tuple in $I_U^n$ that specifies the text region from which $m$ is extracted. Hence, tuples are appended to $I_U^n$ and $O_U^n$ in the order they are generated. After executing $T$ over $P_n$, each IE unit $U$ is associated with two reuse files $I_U^n$ and $O_U^n$ that store intermediate IE results for $U$ for later reuse.

To avoid excessive disk writes caused by individual append operations, we use one block of memory per reuse file to buffer the writes. Whenever a block fills up, we flush the buffered tuples to the end of the corresponding reuse file. The memory overhead during execution is $2|T|$ blocks (one per file), where $|T|$ is the number of IE units in $T$. The I/O overhead, same as the total storage requirement for reuse files, is exactly $\sum_{U \in T}(B(I_U^n) + B(O_U^n))$ blocks, where $B(I_U^n)$ and $B(O_U^n)$ represent the number of blocks occupied by $I_U^n$ and $O_U^n$, respectively. Although it is conceivable for an IE unit to produce more mentions than the size of the input document, in practice the number of mentions is usually no more (and often far smaller) than the input size. Therefore, both the total storage and the I/O overhead are usually bounded by $O(|T|B(P_n))$, where $B(P_n)$ denotes the size of $P_n$ in blocks.

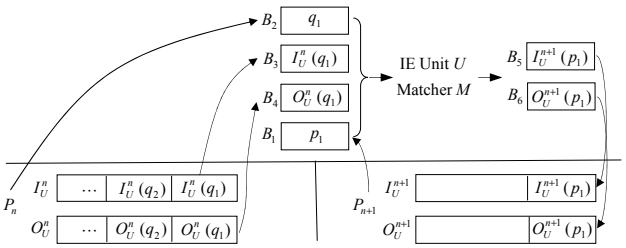


**Figure 4: Movement of data between disk and memory during the execution of IE unit $U$ on page $p_1$.**

# 5. REUSING CAPTURED IE RESULTS

We have described how to capture IE results in reuse files while executing a tree $T$ on snapshot $P_n$. We now describe how to use these results to speed up executing $T$ over the subsequent snapshot $P_{n+1}$.

## 5.1 Scope of Mention Reuse

As discussed earlier, to reuse, we must match each page $p \in P_{n+1}$ with pages in the past snapshots, to find overlapping regions. Many such matching schemes exist. Currently, we match each page $p$ only with the page $q$ in $P_n$ at the same URL as $p$. (If $q$ does not exist then we declare $p$ to have no overlapping regions.) This simplification is based on the observation that pages with the same URL often change relatively slowly across consecutive snapshots, and hence often share much overlapping data. Extending Delex to handle more general matching schemes, such as matching within the same Web site, or matching over all pages of all past snapshots, is an ongoing work.

## 5.2 Overall Processing Algorithm

Within the above reuse scope, we now discuss how to process $P_{n+1}$. Since $P_{n+1}$ can be quite large (tens of thousands or millions of pages), we will scan it only once, and process each page in turn in memory in a streaming fashion.

In particular, to process tree $T$ on a page $p \in P_{n+1}$ (once it has been brought into memory), we need page $q \in P_n$ (the previous snapshot) with the same URL, as well as all intermediate IE results that we have recorded while executing tree $T$ on $q$. These IE results are scattered in various reuse files (Section 4), which can be large and often do not fit into memory. Consequently, we must ensure that in accessing intermediate IE results, we do not probe the reuse files *randomly*. Rather, we want to read them sequentially and access IE results in that fashion.

The above observation led us to the following algorithm. Let $q_1, q_2, \ldots, q_k$ be the order in which we processed pages in $P_n$. That is, we first executed $T$ on $q_1$, then on $q_2$, and so on. The way we wrote reuse files, as described earlier in Section 4, ensures that the IE results in each reuse file are stored in the same order. For example, $I_U^n$ stores all input tuples (for $U$) on page $q_1$ first, then all input tuples on page $q_2$, and so on.

Consequently, we will process pages in $P_{n+1}$ *following the same order*. That is, let $p_i$ be the page with same URL as $q_i$, $i = 1, \ldots, k$. Then we process $p_1$, then $p_2$, and so on. (If a page $p \in P_{n+1}$ does not have a corresponding page in $P_n$, then we can process it at any time, by simply running extraction on it.) By processing in the same order, we only need to scan each reuse file sequentially once.

Figure 4 illustrates the above idea. Suppose we are about to process page $p_1 \in P_{n+1}$. First, we read $p_1$ and $q_1$ into memory (buffers $B_1$ and $B_2$ in the figure).

Next, we execute $T$ on $p_1$ in a bottom-up fashion. Consider the execution tree $T$ in Figure 3.b. We start with executing IE unit $U$. To do so, we bring all intermediate IE results recorded while executing $U$ on $q_1$ (back when we processed $P_n$) into memory. Specifically, let $I_U^n(q_1)$ denote the input tuples for $U$ on page $q_1$. Since $q_1$ is the first page in $P_n$, $I_U^n(q_1)$ must appear at the beginning of file $I_U^n$, and hence can be immediately brought into memory (buffer $B_3$ in Figure 4). Similarly, $O_U^n(q_1)$—the tuples output by $U$ on page $q_1$—must occupy the beginning of file $O_U^n$ and can be immediately read into memory (buffer $B_4$ in Figure 4).

The details of how to execute IE unit $U$ on $p_1$ will be presented next in Section 5.3. Roughly speaking, we identify overlapping regions between $q_1$ and $p_1$, and leverage $I_U^n(q_1)$ and $O_U^n(q_1)$ for reuse. Note that $I_U^n(q_1)$ and $O_U^n(q_1)$ store only the start and end positions of regions in $q_1$, so we need $q_1$ in memory to access these regions. During the execution of $U$ on $p_1$, we produce the input and output tuples of $U$, $I_U^{n+1}(p_1)$ and $O_U^{n+1}(p_1)$, in memory (buffers $B_5$ and $B_6$ in Figure 4, respectively). As described in Section 4, these tuples are also appended to reuse files $I_U^{n+1}$ and $O_U^{n+1}$.

Once we are done with $U$ (for $p_1$), memory reserved for $I_U^n(q_1)$, $O_U^n(q_1)$, and $I_U^{n+1}(p_1)$ can be discarded; however, $O_U^{n+1}(p_1)$ will be retained in memory until it is consumed by the parent operator or IE unit of $U$ in $T$ (in this case, the join operator in Figure 3.b).

Next, we move on to IE unit $V$. We read in $I_V^n(q_1)$ and $O_V^n(q_1)$ from the corresponding reuse files $I_V^n$ and $O_V^n$, and generate $I_V^{n+1}(p_1)$ and $O_V^{n+1}(p_1)$ in memory. Again, once $V$ finishes, only $O_V^{n+1}(p_1)$ needs to stay in memory to provide input to $V$'s parent in $T$. This process continues until we have executed the entire $T$.

Once the entire $T$ finishes execution on $p_1$, we move on to process $T$ on page $p_2$, then $p_3$, and so on. Note that each time we process a page $p_i$, the intermediate IE results of $q_i$ will be at the start of the unread portion of the reuse files, and thus can be read in easily. Consequently, we only have to scan each reuse file once during the entire run over $P_{n+1}$. The total number of I/Os is thus $\sum_{U \in T}(B(I_U^n) + B(O_U^n) + B(I_U^{n+1}) + B(O_U^{n+1})) + B(P_n) + B(P_{n+1})$, i.e., one pass over the current and previous corpus snapshots and all reuse files for the two snapshots. At any point in time (say, when executing IE unit $U$ on page $p_i$), we only need to keep in memory $p_i$, $q_i$, $I_U^n(q_i)$, $O_U^n(q_i)$, $I_U^{n+1}(p_i)$, $O_U^{n+1}(p_i)$, as well as $O_{U'}^{n+1}(p_i)$ for any child $U'$ of $U$. Therefore, the maximum memory requirement for the algorithm (not counting memory needed for buffering writes to reuse files discussed in Section 4, or by the IE units and relational operators themselves) is $O(\max_i(B(p_i)+B(q_i)+(F(T)+1)\max_{U \in T}(B(I_U^n(q_i)), B(O_U^n(q_i)), B(I_U^{n+1}(p_i)), B(O_U^{n+1}(p_i)))))$ blocks, where $F(T)$ is the maximum fan-in of $T$. In practice, under the reasonable assumption that the total size of the extracted mentions is linear in the size of the input page, the memory requirement comes down to $O((F(T)+1)\max_i(B(p_i)+B(q_i)))$.

## 5.3 IE Unit Processing

We now describe in more detail how to execute an IE unit $U$ on a particular page $p$ (in snapshot $P_{n+1}$), whose previous version is $q$ (in snapshot $P_n$). The overall algorithm is depicted in Figure 5.

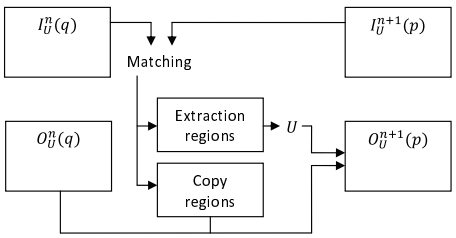


**Figure 5: An illustration of executing an IE unit.**

We start with $I_U^{n+1}(p)$, the set of input tuples to $U$. Each input tuple $(tid, did, s, e, c) \in I_U^{n+1}(p)$ represents a text region $[s, e]$ of page $p$ to which we want to apply $U$, with additional input parameter values $d$. There are two cases. If $U$ has a child in $T$, this set is produced by the execution of the child. If $U$ is a leaf in $T$, which operates directly on page $p$, there is only one input tuple $(did, s, e, c)$, where $did$ is the ID of $p$, $s$ and $e$ are set to 0 and the length of $p$, and $c$ denotes all other input parameters.

To identify reuse opportunities, we consult $I_U^n(q)$, which contains the input tuples to $U$ when it executed on $q$. This set is read in from the reuse file $I_U^n$ as discussed in Section 5.2. Each tuple in $I_U^n(q)$ has the form $(tid', did', s', e', c')$, where $did'$ is the ID of $q$, and $c'$ records the values of additional input parameters that $U$ took when applied to region $[s', e']$ of $q$. To find results to reuse for input tuple $(did, s, e, c) \in I_U^{n+1}(p)$, we "match" the region $[s, e]$ of $p$ with regions of $q$ encoded by tuples in $I_U^n(q)$ with $c' = c$. This matching is done using one of the *matchers* to be described later in Section 5.4 (Section 6 discusses how to select a good matcher).

We repeat the matching step for each input tuple in $I_U^{n+1}(p)$ to find its matching input tuples in $I_U^n(q)$. From the corresponding pairs of matching regions in $p$ and $q$ as well as the scope and context properties of $U$ (Section 3), we derive the *extraction regions* and *copy regions*. Because of space constraint, we do not discuss the derivation process further, but instead refer the reader to [6] for details.

Extraction regions require new work: we run $U$ over these regions of $p$. Copy regions represent reuse. If a copy region is derived from input tuple $(tid', did', s', e', c') \in I_U^n(q)$, we find the joining output tuples (with the same $tid'$) in $O_U^n(q)$. Recall that $O_U^n(q)$ contains the output tuples of $U$ when it executed on $q$, and this set is read in from the reuse file $O_U^n$ as discussed in Section 5.2. The $O_U^n(q)$ tuples with $tid'$ represent the mentions extracted from region $[s', e']$ of $q$, which can be reused by $U$ to produce output tuples for the corresponding copy region.

Regardless of how $U$ produces its output tuples (through reuse or new execution), they are appended to the reuse file $O_U^{n+1}$ (as described in Section 4), and kept in memory until consumed by a parent operator or IE unit in $T$ (as described in Section 5.2).

## 5.4 Identifying Reuse With Matchers

Delex currently employs four matchers—DN, UD, ST, and RU—for matching regions between two pages (more matchers can be easily plugged in as they become available). We describe the first three matchers here only briefly, since they

come from Cyclex. Then, we focus on RU, a novel contribution of Delex that allows sharing the work of matching across IE units.

Given two text regions $R$ (of page $p \in P_{n+1}$) and $S$ (of page $q \in P_n$) to match, DN immediately declares that the two regions have no matching portions, incurring zero running time. Using DN thus amounts to applying IE from scratch to $R$. UD employs a Unix-diff-command like algorithm [24]. It is relatively fast (takes time linear in $|R|+|S|$), but finds only some matching regions. ST is a suffix-tree based matcher, which finds *all matching regions* of $R$ using time linear in $|R| + |S|$. We do not discuss these Cyclex matchers further; see [6] for more details.

The development of RU is based on the observation that we can often avoid repeating much of the matching work for different IE units. This opportunity does not arise in Cyclex because Cyclex considers only a single IE blackbox. To illustrate the idea in a multi-blackbox setting, consider again executing tree $T$ of Figure 3.b on page $p \in P_{n+1}$, and suppose that we execute IE units $U$, $V$, $Y$, and $Z$, in that order. During $U$'s execution we would have matched page $p$ with page $q \in P_n$ with the same URL to find overlapping regions on which we can reuse mentions.

Now consider executing $V$. Here, we would need to match $p$ and $q$ again; clearly, we should take advantage of the matching work we have already performed on behalf of $U$. Next, consider executing $Y$. Here, we often have to match a region $R$ of $p$ with a set of regions $S_1, \ldots, S_k$ of $q$ (as described in Section 5.3) to detect overlapping regions (on which we can reuse mentions produced by $Y$ on page $q$). However, since we have already matched $p$ with $q$ while executing $U$, we should be able to leverage that result to quickly find all overlapping regions between $R$ of $p$ and $S_i$ of $q$.

In general, since all regions to be matched by IE units of an execution tree come from two pages (one from $P_n$ and the other from $P_{n+1}$), and since IE units often match successively smaller regions that are extracted from longer regions (matched by lower IE units), it follows that higher-level IE units can often reuse matching results of lower ones, as described earlier.

We now briefly describe RU, a novel matcher that draws on this idea. While $T$ executes on a page $p$, RU keeps track of all triples $(R, S, \mathcal{O})$, whenever a ST or UD matcher has matched a region $R$ of $p$ with a region $S$ of $q$ and found overlapping regions $\mathcal{O}$. Now suppose an IE unit $X$ calls RU to match two regions $R'$ and $S'$. RU computes the intersection of $R'$ with all recorded $R$ regions, the intersection of $S'$ with all recorded $S$ regions, and then uses these intersections and the recorded overlapping regions $\mathcal{O}$ to quickly compute the set of overlapping regions for $R'$ and $S'$. We omit further details for space reasons.

The four matchers in Delex make different trade-offs between result completeness and runtime efficiency. The next section discusses how Delex assigns appropriate matchers to IE units, thereby selecting a good IE plan.

# 6. SELECTING A GOOD IE PLAN

Given an execution tree $T$, we now discuss how to select appropriate matchers for $T$ using a cost-based approach. We first describe the space of alternatives, then our cost-driven search strategy, and finally the cost model itself.

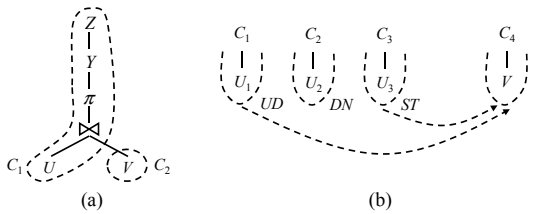


**Figure 6: IE chains and sharing the work of matching across them.**

## 6.1 Space of Alternatives

For each corpus snapshot, we consider assigning a matcher to each IE unit of tree $T$, and then use the so-augmented tree to process pages in the snapshot. Let $|T|$ be the number of IE units in $T$, and $k$ be the number of matchers available to choose (Section 5.4). We would have a total of up to $k^{|T|}$ alternatives. For ease of exposition, we will refer to such an alternative as an *IE plan* whenever there is no ambiguity.

Note that we could make the choice of matchers at even finer levels, such as whenever we must match two regions (while executing $T$ on a page $p$). However, such low-level assignments would produce a vast plan space that is practically unmanageable. Hence, we assign matchers only at the IE-unit level. Even at this level, the plan space is already huge, ranging from 1 million plans for 10 IE units and four possible matchers, to 1 billion plans for 15 IE units, and beyond.

Furthermore, for most plans in this space, optimization is not "decomposable," in that "gluing" the locally optimized subplans together does not necessarily yield a globally optimized plan. The following example illustrates this point.

EXAMPLE 6. *Consider a plan of two IE units $A(B)$, where we apply $A$ to the output of $B$. When optimizing $A$ and $B$ in isolation, we may find that matcher* UD *works best for both. So the best global plan appears to be applying* UD *to both units. However, when optimizing $A(B)$ as whole, we may find that applying* ST *to $A$ and* RU *to $B$ produces a better plan. The reason is that for $A$* ST *may be more expensive (i.e., takes longer to run) than* UD*, but it generates more matching regions, and $B$ can just use* RU *to recycle these regions at a very low cost.*

For the above reasons, we did not look for an exact algorithm that finds the optimal plan. Rather, as a first step, in this paper we develop a greedy solution that can quickly find a good plan in the above huge plan space. We now describe this solution.

## 6.2 Searching for Good Plans

Our solution breaks tree $T$ into smaller pieces, finds a good plan for some initial pieces, and iteratively builds on them to find a good plan to cover other pieces until the entire $T$ is covered. To describe the solution, we start with the concept of *IE chain*:

DEFINITION 6 (IE CHAIN). *An IE chain is a path in tree $T$ such that (a) the path contains a sequence of IE units $A_1, \cdots, A_k$, (b) the path begins with $A_1$ and ends with $A_k$, (c) between each pair of adjacent IE units $A_i$ and $A_{i+1}$, there are no other IE units, and $A_i$ extracts mentions from regions output by $A_{i+1}$, and (d) the chain is maximal in that we can*

**Algorithm 1** Searching for Execution Plan

**Input:** IE execution tree $T$
**Output:** execution plan $G$
$\mathcal{C} \Leftarrow$ partition $T$ //$\mathcal{C}$ is a set of chains
$C_1, \cdots, C_h \Leftarrow$ sort $\mathcal{C}$ in decreasing order of cost estimate
$g_1 \Leftarrow$ findBest($C_1$)
$G \Leftarrow \{g_1\}$
**for** $2 \leq i \leq h$ **do**
  $g_i' \Leftarrow$ findBest($C_i$)
  $B \Leftarrow$ bottom IE units for all chains in $G$
  **if** (any $U \in B$ has the raw data page as input and is assigned ST or UD) **then**
    $g_i'' \Leftarrow$ assign RU to all IE units of $C_i$ reusing the matching results of $U$
    $g_i \Leftarrow$ select $g_i'$ or $g_i''$ with the smaller cost estimate
    $G \Leftarrow G \cup \{g_i\}$
  **else**
    $G \Leftarrow G \cup \{g_i'\}$
  **end if**
**end for**

**Procedure** FindBest($C_i$)

**Input:** chain $C_i = A_1(A_2(\cdots(A_k)\cdots))$
**Output:** best execution plan for $C_i$ in $\mathcal{M}_i$, where $\mathcal{M}_i$ is the set of plans each having at most one IE unit $A_j$, $1 \leq j \leq k$, assigned matcher ST or UD.
$\mathcal{M}_i' \Leftarrow \emptyset$
$g \Leftarrow$ assign DN to each $A_j$, $1 \leq j \leq k$
$\mathcal{M}_i' \Leftarrow \mathcal{M}_i' \cup \{g\}$
**for** $1 \leq j \leq k$ **do**
  $g \Leftarrow$ assign ST to $A_j$, RU to $A_m$, $1 \leq m < j$, and DN to $A_n$, $j < n \leq k$
  $\mathcal{M}_i' \Leftarrow \mathcal{M}_i' \cup \{g\}$
  $g \Leftarrow$ assign UD to $A_j$, RU to $A_m$, $1 \leq m < j$, and DN to $A_n$, $j < n \leq k$
  $\mathcal{M}_i' \Leftarrow \mathcal{M}_i' \cup \{g\}$
**end for**
for each $g \in \mathcal{M}_i'$, estimate its cost using the cost model
return the $g$ with the smallest cost estimate

*not add another IE unit to its beginning or end and obtain another chain satisfying the above properties.*

For example, an IE execution tree $extractTopics(extractAbstract(\overline{d}, abstract))$ is itself a chain because the IE unit $extractAbstract$ extracts abstracts from a document $d$, and then feeds them to IE unit $extractTopics$, which in turn extracts topic strings from the abstract.

Note that the above definition allows two adjacent IE units to be connected indirectly by relational operators that do not belong to any IE units. For example, the chain $C_1$ in Figure 6.a consists of the sequence of IE units $Z$, $Y$, $U$, where $Y$ and $U$ are connected by project-join (and $Y$ extracts mentions from a text region output by $U$).

It is relatively straightforward to partition any execution tree $T$ into a set of IE chains. Figure 6.a shows for example a partition of such a tree into two chains $C_1$ and $C_2$. Note that this is also the only possible partition created by Definition 6, given that $Y$ extracts mentions only from a text region output by $U$ (not from any text region output by $V$). In general, given a tree $T$, Definition 6 creates a unique partition of $T$ into IE chains.

We define the concept of IE chain because, within each chain, it is relatively easy to find a good local plan, as we will see later. Unfortunately, we cannot just find these locally optimal plans *independently*, and then assemble them together to form a good global plan. The reason is that chains can reuse results of other chains, and this reuse often leads to a substantially better plan (than one that does not exploit reuse across chains), as the following example illustrates.

EXAMPLE 7. *Suppose we have found a good plan for chain $C_1$ in Figure 6.a, and this plan applies matcher* ST *for IE unit $U$. That is, for each page $p$ in snapshot $P_{n+1}$, $U$ applies* ST *to match $p$ with $q$, the page with the same URL in $P_n$. Assuming that the running time of matcher* RU *is negligible (which it is in practice), the best local plan for chain $C_2$ is to apply matcher* RU *in IE unit $V$. Since $V$ must also match $p$ and $q$,* RU *will enable $V$ to recycle matching results of $U$, with negligible cost.*

Thus, optimality of IE chains is clearly "interdependent." To take such interdependency into account and yet keep the search still manageable, we start with one initial chain, find a good plan for it in isolation, then extend this plan to cover a next chain, taking into account cross-chain reuse, and so on, until we have covered all chains. Our concrete algorithm is as follows (Algorithm 1 shows the full pseudo code).

**1. Sort the IE Chains:** Using the cost model (see the next subsection), we estimate the cost of each IE chain if extraction were to be performed from the scratch in all IE units of the chain. We then sort the chains in decreasing order of this cost. Without loss of generality, let this order be $C_1, \ldots, C_h$.

**2. Find a Good Plan $g$ for the First Chain:** Since the first chain is the most expensive, we give it the maximum amount of freedom in choosing matchers. To do so, we enumerate the following set of plans for the first chain $C_1$ (based on the heuristics that we explain below):

1. a plan that assigns matcher DN to all IE units of $C_1$;
2. all plans that assign ST to an IE unit $U$ of $C_1$, RU to all "ancestor" IE units of $U$, and DN to all "descendant" IE units of $U$;
3. all plans that assign UD to an IE unit $U$ of $C_1$, RU to all "ancestor" IE units of $U$, and DN to all "descendant" IE units of $U$.

We then use the cost model to select the best plan $g$ from the above set.

Since the cost of RU is negligible in practice (as remarked earlier), it is easy to prove that the above set of plans dominates the set $\mathcal{M}$ of plans where each plan employs matchers ST and UD at most once, i.e., at most one IE unit in the plan is assigned a matcher that is either ST or UD. Thus, the plan we select will be the best plan from $\mathcal{M}$.

We do not examine a larger set of plans because any plan outside $\mathcal{M}$ would contain at least either two ST matchers, or two UD matchers, or an ST matcher together with a UD matcher. Since the cost of these matchers are not negligible, our experiments suggest that plans with two or more such matchers tend to incur high overhead. In particular, they usually underperform plans where we apply just one such expensive matcher relatively early on the chain, and then apply only RU matcher afterward. For this reason, we currently consider only the plan space $\mathcal{M}$.

**3. Extend Plan $g$ to Cover the Second Chain:** First, we repeat the above Step 2 (but replacing $C_1$ with $C_2$), to find a good plan $g'$ for the second chain $C_2$.

Next, let $U$ be the bottom IE unit of chain $C_1$. Suppose the best plan $g$ for $C_1$ assigns either matcher ST or UD to $U$. Then we can potentially reuse the results of this matcher for $C_2$ (if $C_2$ is executed later than $C_1$ in $T$). Hence, we consider

| | |
|---|---|
| $a$ | Average number of input tuples in $I_U$ per page |
| $b$ | Size of $I_U$ on disk (in blocks) |
| $c$ | Size of $O_U$ on disk (in blocks) |
| $d$ | Size of all pages on disk (in blocks) in a snapshot |
| $l$ | Average length of a region encoded by an input tuple |
| $m$ | Number of pages in a single snapshot |
| $v$ | Number of buckets in the in-memory hash table of copy regions |

(a) Meta data statistics

| | |
|---|---|
| $f$ | Fraction of pages with an earlier version in the previous snapshot |
| $s$ | Number of times a matcher is invoked on a region encoded by an input tuple |
| $g$ | After matching region $R$, the ratio of resulting extraction regions to $R$ (in length) |
| $h$ | Number of copy regions generated from matching a region |

(b) Selectivity statistics

**Figure 7: Cost model parameters.**

a reuse-across-chains plan $g''$ that assigns matcher RU to all IE units of $C_2$ (and directing them to reuse from IE unit $U$ of $C_1$).

We then compare the estimated cost of $g'$ and $g''$, and select the cheaper one as the best plan found for chain $C_2$.

**4. Cover the Remaining Chains Similarly:** We then repeat Step 3 to cover the remaining chains. In general, for a chain $C_i$, we could have as many reuse-across-chains plans as the number of chains in the set $\{C_1, \ldots, C_{i-1}\}$ that assign matcher ST or UD to their bottom IE units.

EXAMPLE 8. *Figure 6.b depicts a situation where we have found the best plans for chains $C_1, C_2$, and $C_3$. These plans have assigned matchers* UD, DN, *and* ST *to the bottom IE units $U_1, U_2$, and $U_3$, respectively. Then, when considering chain $C_4$, we will create two reuse-across-chains plans: the first one reuses the results of matcher* UD *of $U_1$, and the second reuses the results of matcher* ST *of $U_3$ (see the figure).*

Once we have covered all the chains, we have found a reasonable plan for execution tree $T$. Our experiments in Section 8 show that such plans prove quite effective on our real-world data sets.

## 6.3 Cost Model

We now describe how to estimate the runtime of an execution plan. Since the difference among all plans is how they execute the IE units of tree $T$ (Section 6.1), we focus on the cost incurred by executing IE units, and ignore other costs. Therefore, we estimate the cost of a plan to be $\sum_{U \in T} t_U$, where $t_U$ denotes the elapsed time of executing the IE unit $U$.

For an IE unit $U$, we further model $t_U$ as the sum of the elapsed time of the steps involved in executing $U$ (Section 5.3). We model the elapsed time of each step as a weighted sum of I/O and CPU costs to capture the elapsed times of highly tuned implementations that overlap I/O with CPU computation (in which case, the dominated cost component will be completely masked and therefore have weight 0) as well as simple implementations that do not exploit parallelism.

To model $t_U$, our cost model employs three categories of parameters. The first category of parameters (listed in Figure 7.(a)) are the meta data of data pages and intermediate results. For these parameters, we use subscript $n$ to represent the value of the parameter on snapshot $n$. For example, $a_n$ denotes the average number of input tuples in $I_U^n(q)$ for a page $q \in P_n$.

The second category of parameters (listed in Figure 7.(b)) are selectivity statistics of a matcher. The last category of parameters are I/O and CPU cost weights $w$, whose subscripts reflect which step incur the associated costs. For all parameters, we use hatted variables to represent parameters are estimated.

We now describe $t_U$ as follows. $t_U$ consists of 4 cost components in executing $U$. The first cost component is the cost of identifying regions encoded by input tuples $(tid, did, s, e, c) \in I_U^{n+1}$ and $(tid', did', s', e', c') \in I_U^n$ where $c = c'$. We model the cost component as:

$$\hat{w}_{1,IO} \cdot b_n + \hat{w}_{1,find} \cdot a_n \cdot \hat{a}_{n+1} \cdot m_{n+1} \cdot \hat{f} \qquad (1)$$

The term $\hat{w}_{1,IO} \cdot b_n$ models the I/O cost of reading in $I_U^n$ into buffer. The term $a_n \cdot \hat{a}_{n+1} \cdot m_{n+1} \cdot \hat{f}$ models the total number of comparisons between arguments $c$ and $c'$ for input tuples in $I_U^n$ and $I_U^{n+1}$ respectively.

The second cost component is the cost of matching the regions identified in the first step. We model this component as:

$$\hat{w}_{2,IO} \cdot d_n \cdot \hat{f} + \hat{w}_{2,mat} \cdot \hat{a}_{n+1} \cdot m_{n+1} \cdot \hat{f} \cdot \hat{s} \cdot \hat{l} \qquad (2)$$

This model accounts for the I/O cost of reading in pages in $P_n$ and the CPU cost of applying matchers. The term $d_n \cdot \hat{f}$ estimates the size (in disk blocks) of raw data pages in $P_n$ that share the same URL, since we only match same URL pages (see Section 5.1). The term $\hat{a}_{n+1} \cdot m_{n+1} \cdot f \cdot s$ estimates the total number of times we apply the matcher when executing $U$ on $P_{n+1}$.

The third cost component is the cost of applying $U$ to all extraction regions. We model this component as:

$$\hat{w}_{3,ex} \cdot (\hat{a}_{n+1} \cdot m_{n+1} \cdot (1 - \hat{f}) \cdot \hat{l} + \hat{a}_{n+1} \cdot m_{n+1} \cdot \hat{f} \cdot \hat{l} \cdot \hat{g}) \qquad (3)$$

We will apply $U$ to those input tuples (in $I_U^{n+1}$) on pages in $P_{n+1}$ that do not have an earlier version in $P_n$. The term $\hat{a}_{n+1} \cdot m_{n+1} \cdot (1 - \hat{f}) \cdot \hat{l}$ estimates the total length of regions encoded in those tuples. In addition, we also need to apply $U$ to the extraction regions on pages $P_{n+1}$ that do have an earlier version in $P_n$. The term $\hat{a}_{n+1} \cdot m_{n+1} \cdot \hat{f} \cdot \hat{l} \cdot \hat{g}$ estimates the length of these extraction regions. In particular, $g$ measures, on average, the fraction of a region we still need to apply $U$ after we match it using a matcher.

The last cost component is the cost of reusing output tuples for copy regions. We model this component as:

$$\hat{w}_{4,IO} \cdot c_n + \hat{w}_{4,copy} \cdot a_n \cdot m_n \cdot \frac{\hat{a}_{n+1} \cdot m_{n+1} \cdot \hat{f} \cdot \hat{h}}{v} \qquad (4)$$

The formula models the I/O cost of reading in $O_U^n$ and the CPU cost of probing the copy regions to determine whether to copy each mention. Delex stores the copy regions in a hash table to facilitate fast lookups. The term $\frac{\hat{a}_{n+1} \cdot m_{n+1} \cdot \hat{f} \cdot \hat{h}}{v}$ estimates the number of hash table entries per bucket.

Notice that we ignore the costs of reading the raw data pages in $P_{n+1}$ and writing out the intermediate results and the final target relation, since these costs are the same for all plans.

Given the cost model, we then estimate the parameters using a small sample $S$ of $P_{n+1}$ as well as the past $k$ snapshots, for a pre-specified $k$. Since our parameter estimation

| Data Sets | DBLife | Wikipedia |
|---|---|---|
| # Data Sources | 980 | 925 |
| Time Between Snapshots | 2 days | 21 days |
| # Snapshots | 15 | 15 |
| Avg # Page per Snapshot | 10155 | 3038 |
| Avg Size per Snapshot | 180M | 35M |

(a) Data sets for our experiments.

| IE Program for DBLife | # IE "Blackboxes" | α (in char.) | β (in char.) |
|---|---|---|---|
| talk (speaker, topics) | 1 | 155 | 9 |
| chair (person, chairType, conference) | 3 | 9458 | 5 |
| advise (advisor, advisee, topics) | 5 | 20539 | 12 |

| IE Program for Wikipedia | # IE "Blackboxes" | α (in char.) | β (in char.) |
|---|---|---|---|
| blockbuster (movie) | 2 | 10625 | 7 |
| play (actor, movie) | 4 | 22705 | 7 |
| award (actor, movie, role, award ) | 6 | 30506 | 15 |

(b) IE programs for our experiments.

**Figure 8: Data sets and IE programs for our experiments**

techniques are similar to those in Cyclex, we do not discuss the details any further.

# 7. PUTTING IT ALL TOGETHER

We now describe the end-to-end Delex solution. Given an IE program $\mathcal{P}$ written in xlog, we first employ the techniques described in [28] to translate and optimizes $\mathcal{P}$ into an execution tree $T$, and then pass $T$ to Delex.

Given a corpus snapshot $P_{n+1}$, Delex first employs the optimization technique described in Section 6 to assign matchers to the IE units of $T$. Next, Delex executes the so-augmented tree $T$ on $P_{n+1}$, employing the reuse algorithm described in Section 5 and the reuse files it produced for snapshot $P_n$. During execution, it captures and stores intermediate IE results (for reuse in the subsequent snapshot $P_{n+2}$), as described in Section 4.

Note that Delex executes essentially the same plan tree $T$ on all snapshots. The only aspect of the plan that changes across snapshots is the matchers assigned to the IE units. Our experiments in Section 8 show that for our real-world data sets this scheme already performs far better than current solutions (e.g., applying IE from scratch, running Cyclex, reusing IE results on duplicate pages). Exploring more complex schemes, such as re-optimizing the IE program $\mathcal{P}$ for each snapshot or re-assigning the matchers for different pages, is a subject of ongoing work. The following theorem states the correctness of Delex:

THEOREM 1 (CORRECTNESS OF Delex). *Let $M_{n+1}$ be mentions of the target relation $R$ obtained by applying IE program $\mathcal{P}$ from scratch to snapshot $P_{n+1}$. Then* Delex *is correct in that when applied to $P_{n+1}$ it produces exactly $M_{n+1}$.*

# 8. EMPIRICAL EVALUATION

We now empirically evaluate the utility of Delex. Figure 8 describes two real-world data sets and six IE programs used in our experiments. DBLife consists of 15 snapshots from the DBLife system [13], and Wikipedia consists of 15 snapshots from Wikipedia.com (Figure 8.a). The three DBLife IE programs extract mentions of academic entities and their relationships, and the three Wikipedia IE programs extract mentions of entertainment entities and relationships (Figure 8.b). Figure 9 shows for example the execution plan

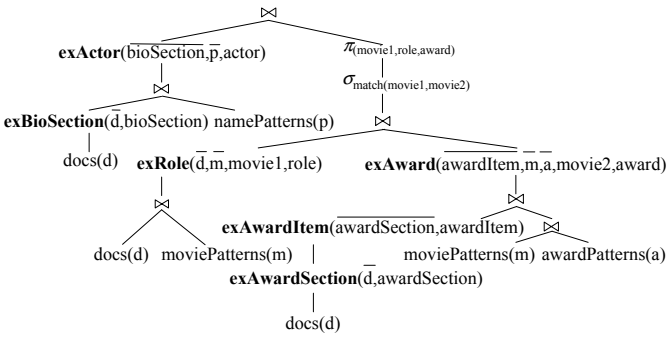


**Figure 9: The execution plan used in our experiments for the "award" IE task.**

used in our experiments for the "award" IE task (with IE blackboxes shown in bold font). The above IE programs are rule-based. However, we also experimented with an IE program consisting of multiple learning-based blackboxes, as detailed at the end of this section.

We obtained the scope $\alpha$ and context $\beta$ of each IE blackbox and the entire IE program by analyzing the IE blackboxes and their relationships. The technical report [7] describes this analysis in details.

**Runtime Comparison:** For each of the six IE tasks in Figure 8.b, Figure 10 shows the runtime of Delex vs. that of other possible baseline solutions over all consecutive snapshots. We consider three baselines: No-reuse, Shortcut, and Cyclex. No-reuse re-executes the IE program over all pages in a snapshot; Shortcut detects identical pages, then reuses IE results on those; and Cyclex treats the whole IE program as a single IE blackbox.

On DBLife, No-reuse incurred much more time than the other solutions. Hence, to clearly show the differences in the runtimes of all solutions, we only plot the runtime curves of Shortcut, Cyclex, and Delex on DBLife (the left side of Figure 10). Since in each snapshot both Cyclex and Delex employ a cost model to select and execute a plan, their runtime includes statistic collection, optimization, and execution times.

Figure 10 shows that, in all cases, No-reuse (i.e., rerunning IE from the scratch) incurs large runtimes, while Shortcut shows mixed performance. On DBLife, where 96-98% of pages remain identical on consecutive snapshots, it performs far better than No-reuse. But on Wikipedia, where many pages tend to change (only 8-20% pages remain identical on consecutive snapshots), Shortcut is only marginally better than No-reuse. In all cases, Cyclex performs comparably or significantly better than Shortcut.

Delex however outperforms all of the above solutions. For "talk" task, where the IE program contains a single IE blackbox, Delex performs as well as Cyclex. For all the remaining tasks, where the IE program contains multiple IE blackboxes, Delex significantly outperforms Cyclex, cutting runtime by 50-71%. These results suggest that Delex was able to exploit the compositional nature of multi-blackbox IE programs to enable more reuse, thereby significantly speeding up program execution.

**Contributions of Components:** Figure 11 shows the runtime decomposition of the above solutions (numbers in the figure are averaged over five random snapshots per IE task). "Match" is the total time of applying all matchers in

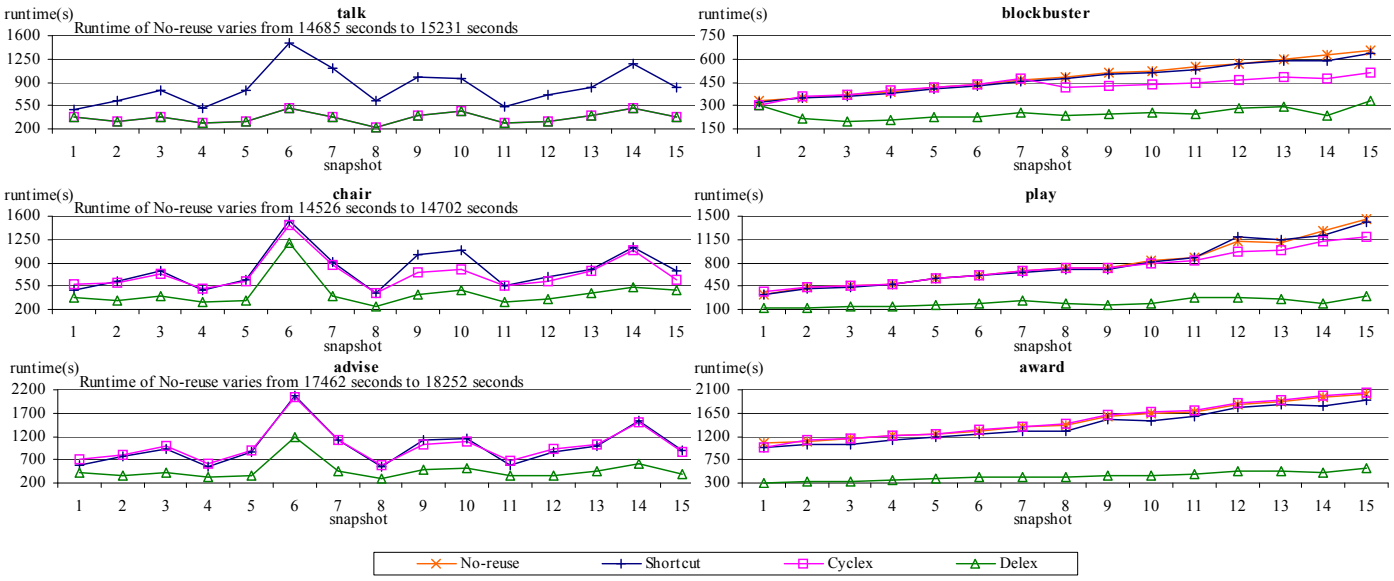


**Figure 10: Runtime of No-reuse, Shortcut, Cyclex, and Delex.**

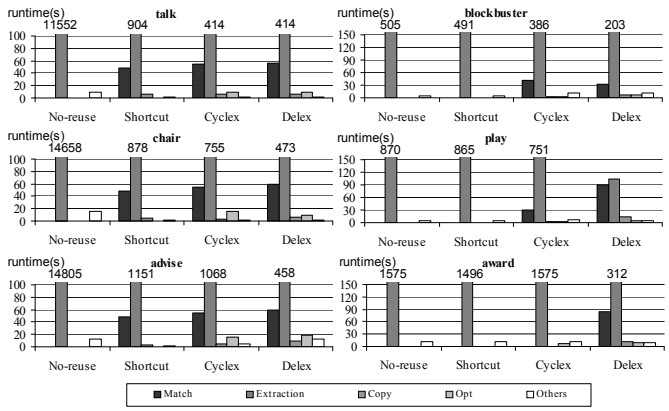


**Figure 11: Runtime decomposition of No-reuse, Shortcut, Cyclex and Delex.**

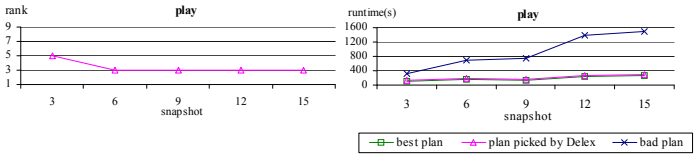


**Figure 12: Performance of the optimizer.**

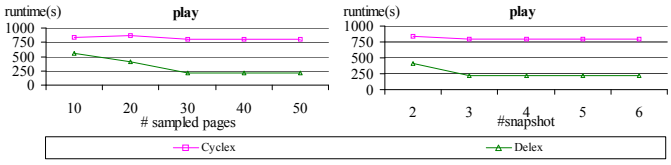


**Figure 13: Sensitivity analysis.**

the execution tree. "Extraction" is the total time to apply all IE extractors. "Copy" is the total time to copy mentions. "Opt" is the optimization time of Cyclex and Delex. Finally, "Others" is the remaining time (to apply relational operators, read file indices, etc.).

The results show that matching and extracting dominate runtimes. Hence we should focus on optimizing these components, as we do in Delex. Furthermore, Delex spends more time on matching and copying than Cyclex and Shortcut in complex IE programs (e.g., "play" and "award"). However, this effort clearly pays off (e.g., reducing the extraction time by 37-85%). Finally, the results show Delex incurs insignificant overhead (optimization, copying, etc.) compared to its overall runtime.

We also found that in certain cases the best plan (one that incurs the least amount of time) employs RU matchers, and that the optimizer indeed selected such plans (e.g., for "chair" and "advise" IE tasks), thereby significantly cutting runtime (see the left side of Figure 10). This suggests that reusing across IE units can be highly beneficial in our Delex context.

**Effectiveness of the Delex Optimizer:** To evaluate the Delex optimizer, we enumerate all possible plans in the plan space, and then compare the runtimes of the best plan versus the one selected by the optimizer. To conduct the experiment, we first selected the "play" IE task, whose plan space contains 256 plans, thereby enabling us to enumerate and run all plans. We then ranked the plans in increasing order of their actual runtimes. Figure 12.a shows the positions in this ranking for the plan selected by the optimizer, over five snapshots. The results show that the optimizer consistently selected a good plan (ranked number five or three). Figure 12.b shows the runtime of the actual best plan, the selected plan, and the worst plan, again over the same five snapshots. The results show that the selected plan performs quite comparably to the best plan, and that optimization is important, given the significantly varying runtimes of the plans.

**Sensitivity Analysis:** Next, we examined the sensitivity of Delex with respect to the main input parameters: number

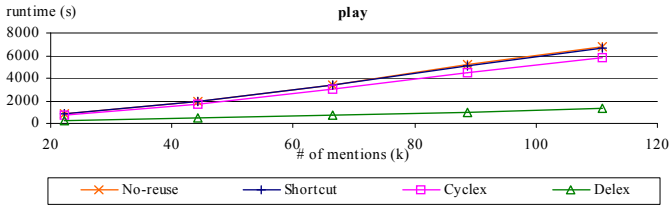


**Figure 14: Runtime comparison wrt number of mentions.**

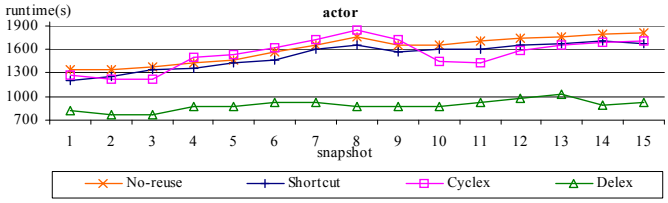


**Figure 15: Runtime comparison on a learning based IE program.**

of snapshots, size of sample used in statistics estimation, and the scope and context values.

Figure 13.a plots the runtime of the plans selected by the optimizers of Delex and Cyclex as a function of sample size, only for "play" (results for other IE tasks show similar phenomenons). Figure 13.b plots the runtime of the plans selected by the optimizer of Delex and Cyclex as a function of the number of snapshots.

The results show that in both cases Delex only needs a few recent snapshots (3) and a small sample size (30 pages) to do well. Furthermore, even when using statistics over only the last 2 snapshots, and a sample size of 10 pages, Delex can already reduce the runtime of Cyclex by 25%. This suggests that while collecting statistics is crucial for optimization, we can do so with a relatively small number of samples over very recent snapshots.

We also conducted experiments to examine the sensitivity of Delex with respect to the $\alpha$ and $\beta$ of IE "blackboxes" (figure omitted for space reasons). We found that the runtime of Delex grows gracefully when $\alpha$ and $\beta$ of IE "blackboxes" increase. Consider for example a scenario in our experiments: randomly selecting an IE blackbox in the "play" task and increasing its $\alpha$ and $\beta$ to examine the change in Delex's time. When we increased $\alpha$ from 52 to 150, the averaged runtime of Delex over five randomly selected snapshots only increases by 15% (from 216 seconds to 248 seconds). When we further increased $\alpha$ to 250 (five times of the original $\alpha$), the averaged runtime of Delex over the same five snapshots increases by only 38% (from 216 seconds to 298 seconds). We observe a similar phenomenon for $\beta$. The results suggest that a rough estimation of the $\alpha$ and $\beta$ of the IE blackboxes does increase the runtime of Delex, but in a graceful fashion.

**Impact of Capturing IE Results:** We also evaluated the impact of capturing IE results on Delex. To do so, we varied the number of mentions extracted by the IE blackboxes and then examined the runtimes of Delex and the baseline solutions. For example, given the IE program "play," we changed the code of each IE blackbox in "play" so that a mention extracted by the IE blackbox is output multiple times. Then we applied Delex and the baseline solutions to this revised IE program of "play." Figure 14 plots these runtimes on "play" as a function of the total number of mentions extracted by all IE blackboxes.

The results show Delex continues to outperform the baseline solutions by large margins as the total number of mentions grows. This suggests that Delex scales well with respect to the number of extracted mentions (and thus the size of captured IE results). Furthermore, we found that as the number of mentions grows by 400% (from 22K to 110K), the time Delex spends on capturing and reusing the IE results only grows by 88% (from 17 seconds to 32 seconds). Additionally, the overhead of capturing and reusing IE results incurred by Delex remains to occupy an insignificant portion (3% - 8%) of its overall runtime. This suggests that the overhead of capturing IE results does increase as the number of extracted mentions increases, but only in a graceful manner.

**Learning-based IE Programs:** Finally, we wanted to know how well Delex works on IE programs that contain learning-based IE blackboxes. To this end, we experimented with an IE program proposed by a recent work [30] to automatically construct infoboxes (tabular summaries of an object's key attributes) in Wikipedia pages. This IE program extracts name, birth name, birth date, and notable roles for each actor. To do this, it employs a maximal entropy (ME) classifier to segment a raw data page into sentences, then employs four conditional random field (CRF) models – one for each attribute – to extract the appropriate values from each of the sentences.

To apply Delex, we first converted the above IE program into an xlog program that consists of five IE blackboxes. These blackboxes capture the ME classifier and the four CRF models, respectively. Then we derived $\alpha$ and $\beta$ for each of the blackboxes. For example, given a delimit character in a raw data page, the ME classifier examines its context (i.e., surrounding characters) to determine if the delimit character is indeed the end of a sentence. Given this, we can set $\alpha_{ME}$ to be the maximal number of characters in a sentence, and $\beta_{ME}$ to be the maximal number of characters in the contexts examined by the ME classifier (321 and 16 in our experiment, respectively). It is more difficult to derive tight values of $\alpha_{CRF}$ and $\beta_{CRF}$ for the four CRF models, as these models are quite complex. However, we can always set them to the length of the CRF model's longest input string, i.e., the longest sentence, and this is what we did in the current experiment.

Figure 15 shows the runtime of Delex and the three baseline solutions on the above xlog program running on Wikipedia. The results show that both Shortcut and Cyclex only perform marginally better than No-reuse, due to significant change of pages across snapshots and large $\alpha$ (17824 characters) of the entire IE program. However, Delex significantly outperforms all three solutions. In particular, Delex reduces the runtime of Cyclex by 42-53%. This suggests that Delex can benefit from exploiting the compositional nature of multi-blackbox learning-based IE programs, even though we are not able to derive tight $\alpha$ and $\beta$ for some learning-based IE blackboxes (e.g. the complex CRF models) in the programs.

## 9. CONCLUSIONS AND FUTURE WORK

A growing number of real-world applications involve IE over dynamic text corpora. Recent work on Cyclex has shown that executing such IE in a straightforward manner

is very expensive, and that recycling past IE results can lead to significant performance improvements. Cyclex, however, is limited in that it handles only IE programs that contain a single IE blackbox. Real-world IE programs, in contrast, often contain *multiple IE blackboxes* connected in a workflow.

To address the above problem, we have developed Delex, a solution for effectively executing multi-blackbox IE programs over evolving text data. As far as we know, Delex is the first in-depth solution for this important problem. It opens up several interesting directions that we are planning to pursue. These include (a) how to efficiently match a page with all past pages, so that we can expand the scope of reuse, (b) how to handle extractors that extract mentions across multiple pages, and (c) how to handle IE programs that contain recursion and negation.

**Acknowledgments:** We thank Luis Gravano for invaluable comments. This work is supported by NSF Career grant IIS-0347943, and grants from IBM, Yahoo, Microsoft, and Google to the third author; NSF Career grant IIS-0238386 and IIS-0713498 to the fourth author; and NSF grant ITR IIS-0326328 to the fifth author. The research was done while the second author was in the University of Wisconsin.

## 10. REFERENCES

[1] `http://langrid.nict.go.jp`.

[2] E. Agichtein and S. Sarawagi. Scalable information extraction and integration (tutorial). *KDD-06*.

[3] K. Beyer, V. Ercegovac, R. Krishnamurthy, S. Raghavan, J. Rao, F. Reiss, E. J. Shekita, D. Simmen, S. Tata, S. Vaithyanathan, and H. Zhu. Towards a scalable enterprise content analytics platform. *IEEE Data Eng. Bull.*, 32(1):28–35, 2009.

[4] B. Bhattacharjee, V. Ercegovac, J. Glider, R. Golding, G. Lohman, V. Markl, H. Pirahesh, J. Rao, R. Rees, F. Reiss, E. Shekita, and G. Swart. Impliance: A next generation information management appliance. *CIDR-07*.

[5] Y. Cai, X. L. Dong, A. Y. Halevy, J. M. Liu, and J. Madhavan. Personal information management with semex. *SIGMOD-05*.

[6] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan. Efficient information extraction over evolving text data. *ICDE-08*.

[7] F. Chen, B. J. Gao, A. Doan, J. Yang, and R. Ramakrishnan. Optimizing complex extraction programs over evolving text data. *Technical report, UW-Madison, 2009*. Available at `http://www.cs.wisc.edu/~fchen/delex-tr.pdf`.

[8] J. Cho and H. Garcia-Molina. Effective page refresh policies for web crawlers. *TODS-03*.

[9] E. Chu, A. Baid, T. Chen, A. Doan, and J. Naughton. A relational approach to incrementally extracting and querying structure in unstructured data. *VLDB-07*.

[10] W. Cohen and A. McCallum. Information extraction from the world wide web(tutorial). *KDD-03*.

[11] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. *ACL-02*.

[12] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building structured web community portals: A top-down, compositional, and incremental approach. *VLDB-07*.

[13] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan. DBLife: A community information management platform for the database research community (demo). *CIDR-07*.

[14] A. Doan, L. Gravano, R. Ramakrishnan, and S. Vaithyanathan. Special issue on managing information extraction. *SIGMOD Rec.*, 37(4), 2008.

[15] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing information extraction: state of the art and research directions (tutorial). *SIGMOD-06*.

[16] J. Edwards, K. McCurley, and J. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. *WWW-01*.

[17] D. Ferrucci and A. Lally. UIMA: An architectural approach to unstructured information processing in the corporate research envrionment. *Nat. Lang. Eng.*, 10(3-4):327–348, 2004.

[18] A. Gupta and I. Mumick. *Materialized Views: Techniques, Implementations and Applications*. MIT Press, 1999.

[19] M. Herscovici, R. Lempel, and S. Yogev. Efficient indexing of versioned document sequences. *ECIR-07*.

[20] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. To search or to crawl? Towards a query optimizer for text-centric tasks. *SIGMOD-06*.

[21] A. Jain, A. Doan, and L. Gravano. SQL queries over unstructured text batabases. *ICDE-07*.

[22] G. Kasneci, M. Ramanath, F. Suchanek, and G. Weikum. The YAGO-NAGA approach to knowledge discovery. *SIGMOD Rec.*, 37(4):41–47, 2008.

[23] L. Lim, M. Wang, J. Vitter, and R. Agarwal. Dynamic maintenance of web indexes using landmarks. *WWW-03*.

[24] E. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–256, 1986.

[25] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. *ICDE-08*.

[26] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

[27] S. Satpal and S. Sarawagi. Domain adaptation of conditional probability models via feature subsetting. *ECML/PKDD-07*.

[28] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. *VLDB-07*.

[29] D. S. Weld, F. Wu, E. Adar, S. Amershi, J. Fogarty, R. Hoffmann, K. Patel, and M. Skinner. Intelligence in wikipedia. *AAAI-08*.

[30] F. Wu and D. S. Weld. Autonomously semantifying wikipedia. *CIKM-07*.

[31] J. Zhang and T. Suel. Efficient search in large textual collections with redundancy. *WWW-07*.